

An Approach to Scheduling Task Graphs with Contention in Communication

by

Joseph B. Collins

June 4, 2001

The Processing Graph Method Tool (PGMT) product is being released under the GNU General Public License Version 2, June 1991 and related documentation under the GNU Free Documentation License Version 1.1, March 2000.

<http://www.gnu.org/licenses/gpl.html>

ABSTRACT

Algorithms that schedule task graphs for execution on parallel processors frequently use simplifying assumptions about the behavior and performance of networks that pass data between tasks. When there is a high communications-to-computations ratio for a parallel application some of these assumptions may fail. The result is that the network may become a bottleneck in executing the application and the scheduling algorithm may not be sufficiently valid to be useful. We examine this problem for scheduling real-time applications and draw some conclusions on how to do scheduling for general parallel processing. We conclude that whenever communications tasks are a significant part of the workload they must be scheduled, much as processing tasks are, and that the network resources must be allocated to perform communications tasks, much as processors are allocated to perform processing tasks. Moreover, we conclude that due to variations in network design and behavior, a scheduling method useful for different machines must use a network simulator, different for each network. We describe how to represent communications tasks in task graphs and consider how to do scheduling using a network simulator.

INTRODUCTION: TASK GRAPHS, MESSAGE PASSING, AND SCHEDULING

Scheduling task graphs for parallel processing is a problem encountered in compiling parallelizable applications for execution. By the phrase “task graph” we mean a Directed Acyclic Graph (DAG) where computational tasks are represented by nodes and directed arcs represent data dependencies between nodes. The structure of the task graph has the information required to determine whether computational activities are independent or dependent. This allows consideration of ways to schedule concurrent and successive computational activity. In addition to arcs and nodes task graphs frequently have costs, related to the time needed to progress along a path in the graph, associated with nodes, or edges, or both.

We assume a multiple instruction stream, multiple data stream (MIMD), message-passing architecture as our model of parallel processing. Frequently in our discussion we use the term “network”. By “network” we mean any resource that moves a message between different computational contexts. We note that this model does not exclude shared memory machines since message passing may be simulated on a shared memory machine.

A schedule for executing a task graph on a parallel computer is a mapping of task sequences to processors, one task sequence for each processor used. The sequence for a given processor determines the order in which tasks are to be executed on that processor. We assume that the schedule is static and nonpreemptive: computational tasks are executed to completion without interrupting each other. Each schedule sequence may be executed concurrently and each task is enabled to execute when all input data is available and prior computational tasks in its schedule sequence have finished. All of the tasks in the task graph must be included in the schedule for the schedule to be complete. If any additional tasks are required in order to execute the scheduled task graph on a specific parallel computer, they too must be in a complete schedule. (We will show that communication tasks are this kind of task). Finally, while the use of null tasks in the schedule may decrease the execution time for completion of the task graph, we will not discuss them here.

For scheduling to be useful, various pieces of execution time performance information are needed. Of greatest importance is the communications to computation ratio (CCR), the ratio of communication time to computation time. Before scheduling, the CCR may be bounded from above for the whole task graph by dividing the sum of all possible communication times (edge costs) in the task graph by the sum of task execution times. CCR bounds may also be computed for any subgraph of the task graph to be scheduled. The CCR bound is an

indicator in deciding whether or not it is advantageous to distribute a task graph. If a CCR bound is very large, then it may be that distributing the corresponding task graph onto multiple processors is worse than not distributing it. If some of the communications have low cost then it may be that some partition of the task graph can be gainfully distributed. If the CCR bound is vanishingly small, then the number of processors we can gainfully use is limited only by the number of tasks that can be simultaneously executed, as is manifest in the task graph. Actual CCRs may only be computed as a function of a scheduled task graph and accurate computation of a CCR requires a machine performance model that includes a network behavioral model.

Signal Processing Scheduling Requirements

We are interested in developing task graphs for real-time execution. A principal requirement of real-time execution is that a schedule must meet specified latency constraints. For many non-real-time applications there is no defined performance failure for slow applications: good performance means high throughput, and the quality of an application's performance is proportional to its speed. Real-time applications, on the other hand, succeed if they meet specified latency constraints and fail otherwise. Consequently, we are interested in knowing the maximum time required to execute both individual tasks and the task graph. Our scheduling objective function may be taken as minimizing the maximum execution time of the task graph.

We need to schedule signal processing task graphs. Signal processing applications frequently have the property that individual tasks have predictable execution time performance statistics and data inputs and outputs have predictable sizes. These characteristics may not be available a priori for arbitrary applications, but we assume they can be measured and predicted. We also note that signal processing applications are frequently communications intensive to the point that the network becomes the bottleneck. This happens due to a combination of high CCR and network contention. When this occurs, accuracy in predicting network behavior, including contention, is necessary for obtaining good schedules.

Finally, we want to schedule task graphs for execution on different parallel computational platforms. Building a scheduler that works for many different computational platforms is a challenge. To do this, we need to allow for unanticipated behaviors of arbitrary networks.

RECENT NETWORK MODELS AND SCHEDULING ALGORITHMS

General Network Models

Some effort has been made to model the message passing performance of general networks. One model of network performance is the LogGP model [1], an elaboration of the LogP model [2]. The LogGP model uses six parameters: latency, processor overhead, available per processor bandwidth for short messages, available per processor bandwidth for long messages, a size limit for short messages, and the number of processors. These parameters are fitted with respect to a particular system configuration. While the model does not incorporate network contention, messages are queued and serially accepted on a single port to exit or enter a processor. At any given time a processor can either be sending or receiving a single message.

Another model of network behavior is the distributed random access machine (DRAM) model [6]. In the DRAM model there are N processors and each binary partition, or cut, of these N processors has a specified transmission capacity, "equal to the number of wires", between the two processor subsets. A complete DRAM specification specifies a capacity constraint for each binary partition. A basic assumption of the DRAM model is that for a set of messages, the time required to send data across the cut is proportional to the load divided by the capacity. This is the minimum possible time given the capacity constraint and the

uniform allocation of network resources to messages. An underlying assumption is that the network routing protocols will be designed to make optimum use the available capacity.

The LogGP model is used to predict mean performance times with no network contention. Its parameters are statistically estimated from test data. The DRAM model predicts the minimum transmission rates given capacity constraints. Some algorithms have been analyzed to demonstrate the validity of these models. Each model uses a specific assumption about the queuing dynamics. Neither of these models have a general representation for the queuing behavior of networks. Below we will review some of the assumptions made in the literature about the effects communications tasks have on scheduling and we will discuss how these particular models, LogGP and DRAM, may play a role in better schedulers.

Scheduling

A good schedule has short execution time. The basic problem in scheduling task graphs for parallel computing is efficiently finding a good schedule for an arbitrary task graph under specified constraints. Examples of constraints are limits on the number of processors used and bounds on network bandwidth. Non-trivial representations of the scheduling problem are NP-complete [10], so efficient scheduling methods are usually heuristic. Various heuristics have been developed based on critical path methods and clustering methods [3, 4, 8]. Much of this literature concerning the scheduling of a task graph for execution on a parallel-processing machine makes simplifying assumptions about how communication tasks are handled. Simplifying assumptions are made because they make the problem more tractable for heuristics, even though they do not change the essential computational complexity of finding the optimal solution to the resulting scheduling problem. One must be careful not to draw conclusions that depend on side effects of these simplifying assumptions.

As an example, one simplifying assumption is that the time it takes to pass a message is zero if the communicating tasks are on the same processor and a predetermined delay if the succeeding task is on a different processor. This assumption has the advantage that there is no change in the task graph structure conditioned on the existence of message passing: only a reassignment of edge costs is required. Another common assumption is that communications tasks do not consume processor resources. This assumption implies the use of dedicated devices that rout messages without interfering with the processing of computational tasks, e.g., DMA (Direct Memory Access) devices. A side effect consequence of these assumptions is that communications tasks are not assumed to be queued in any way on the same processor. Since interprocessor communications are represented as only a fixed delay, another consequence is that no resources, such as a network or bus, are ever tied up. These side effects amount to an assumption of no contention in message passing. Conclusions drawn from a contention free model are probably only useful as the CCR tends to be small. There are some scheduling algorithms that do recognize network contention [9], but, to be tractable, a specific scheduling algorithm must usually be based on a simple, specific network model, such as a multichannel bus.

We will show that neglecting contention when scheduling task graphs that have high CCRs is only useful in the context of a highly connected network connecting processors that each have many asynchronous communication ports. In addition, we believe that to build schedulers with broad applicability it is preferable to find a single framework that accounts for the essential properties of most networks. Accordingly, we sketch out such a framework.

Required Features of a Network Model

We identify below features that would be required in a faithful model of network behavior:

- (a) Linearity: For a given message the minimum time required to pass the message is a sum of terms proportional to the size of the message, with constant terms that incorporate processor overhead and network latency.

Linearity, a basic feature, is reflected in the LogGP model. The message-passing task is partitioned into subtasks performed by the network and each processor. Linearity captures the contention-free behavior of a network passing a point to point message. For a given throughput, the time to move data is a linear function of its size. The startup, latency, and completion times of message passing are independent of message size.

- (b) Limited I/O port capacity: For multiple messages being simultaneously sent and received on a given processor there exist rate limitations on the overall processor input and output (I/O).

We will call the means by which data gets onto or off a processor the I/O ports. These have static capacity limits and, in the context of multiple messages, dynamic rate limits. For example, when a task is completed that has multiple immediate dependent tasks on other processors, there is a temporary glut in communications resource demand. The I/O ports of a processor may not be able to handle its outstanding message-passing requests expeditiously. There are two cases that usually occur: either all messages will get immediate routing at less than average throughput, or some messages will get immediate routing and others will be delayed. Either of these cases can be implemented in a variety of ways. Messages may be sent by circuit switching, with possible interruption, or packet switching. In either case, there may be use of priorities. Message sending tasks on processors may be explicitly scheduled, or they may be called concurrently and processed asynchronously.

The details of how message tasks are sent off processor makes a significant performance difference. For example, take a task, which just having completed now enables two messages to be sent to other processors to enable further task execution on the foreign processors. Assume that there is only one port by which to exit the processor, and assume that data is sent at a fixed rate. Also, assume that in the context of no other messages being simultaneously passed, each message would take ten time units. Now, sending both messages off processor will not complete until twenty time units have passed. If a small part of each message were passed at a time, alternating between the two messages, neither message would be completely passed until approximately twenty time units had elapsed. On the other hand, if messages were passed in a strict serial manner, then one message would be passed at the end of ten time units and the second would be completely passed at the end of twenty time units. The different completion times may be taken advantage of, for example, by delaying the initiation of one of the message passing tasks not on a critical path.

- (c) Limited network capacity: There are rate limitations inherent in the network structure that limits the instantaneous message traffic flow.

Network capacity limits consist of two parts, one that is static and one that is dynamic. The static limitation is captured, for example, by the capacity constraints of the DRAM model. The dynamic part results from queuing. The time it takes to pass a given message is functionally dependent on the background message traffic. The background message traffic is dependent on the overall communication required at that point in time in the execution schedule for the task graph. For real-time, we want to determine an upper bound on the execution time, therefore we cannot make the optimistic assumption that the network protocols route messages in the least time allowed by the capacity constraints, as in the DRAM model. This is because queuing, while necessary to increase throughput of a loaded system, will tend to increase latency as well.

WHY MODELING CONTENTION IS NECESSARY

We now examine the behavior of a network with only the first feature, (a) above, assuming no contention in message passing. We first define a characteristic of task graphs, a degree of concurrency [3], that bounds the number of processors that will be useful for executing the task graph. Assume that a task graph is represented by a Directed Acyclic Graph (DAG), where compute tasks are represented by nodes of the DAG and data dependencies are represented by directed arcs. The DAG, by transitive closure, also represents a partial order, a relation we will call dependency. If one task depends on another either immediately or by multiple traversals along directed arcs, then that task pair is in the partial order relation. If a task pair is not in the partial order relation, then we will call them not dependent. If two tasks are not dependent, they may be executed simultaneously; if one is dependent on the other, they may not be executed simultaneously. We may find many sets of tasks in the task graph where each element of a given set is not dependent on any other element of the same set. One or more of these sets may have more elements than the other sets, i.e.; there is a maximal cardinality over the sets of mutually non-dependent tasks. This maximal cardinality represents the maximum number of tasks that may be executed simultaneously, which we call C , the degree of concurrency of the task graph.

Concurrency is related to scalability. Let us say we have a class of graphs representing an application, where each graph operates on different size data input. For the application to be scalable, we must be able to use proportionately more processors as the data input size grows in order to execute the application in fixed time. If the maximum number of processors that can be utilized is limited to the degree of concurrency of the task graph, the concurrency must increase as the application is scaled up.

Contention in communication affects scalability. As an illustrative example, in Fig. (1) we consider how a task graph having $(T + 2)$ tasks may be scheduled. Assume that all tasks take one time unit to execute and that all sending tasks also take one time unit, but do not consume processor CPU-time. The CCR is bounded above by $2/(1+2/T)$. The tables show one and two processor schedules for the case where $T = 3$. Executing all five tasks on processor P1 takes five time units (left hand table, Fig (1)). A different schedule, which executes one of the tasks, C3, on processor P2 also takes five time units (right hand table, Fig. (1)). The network must execute a send from task A to task C3 and from task C3 to task B. If we assume an unlimited number of processors and unlimited communications channels, then for arbitrary T we can effectively use $(T - 2)$ processors and complete the task graph in five time units. To do this we will need $(T - 1)$ communications channels connected to processor P1.

On the other hand, assume there is only one channel to handle the communications tasks and all of the communications must be queued. If we schedule tasks A and B to processor P1 then every task not performed on processor P1 will require two messages to be sent, during which time processor P1 can execute two of the tasks $\{C_i\}$. This allows one third of the tasks to be offloaded and executed on another processor. For arbitrarily large T we will only be able to effectively use two processors and it will take approximately $2/3 * T$ time units to complete the task graph. Finally, if we do use a schedule using $(T - 2)$ processors where tasks A and B are scheduled to processor P1 and messages are queued on one port of processor P1, the schedule will take approximately $2 * T$ time units. This is worse than executing all tasks on a single processor.

The result of making the assumption that communication bandwidth scales as the number of processors is that our application appears perfectly scalable. (By perfectly scalable we mean that increasing the number of tasks and the number of processors by the same factor results in the same execution time). On the other hand, either of the contention assumptions allows utilization of only a fixed number of processors, i.e., the application is not scalable. The execution time is proportional to the number of tasks. Consider what is happening. A good schedule for a task graph will take advantage of the concurrency in the task graph, assigning concurrent tasks to different processors. In order to take advantage of concurrency in a task

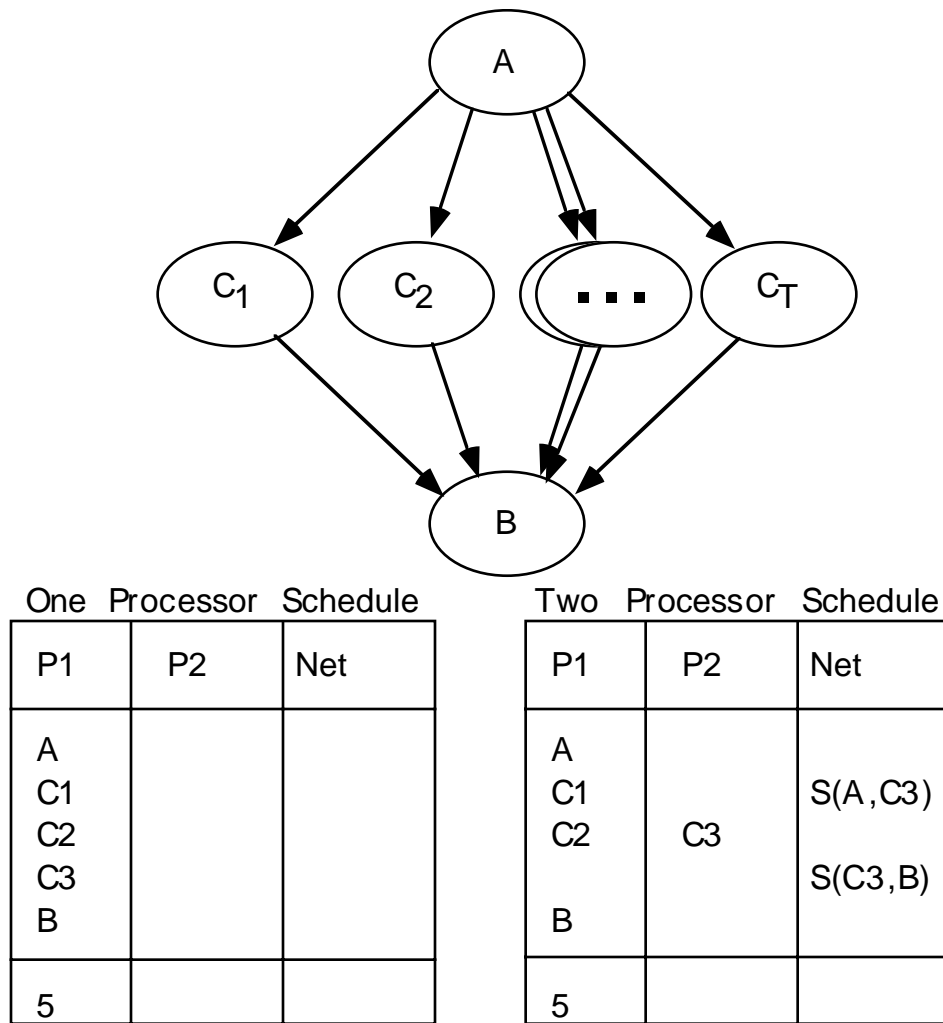


Figure 1

Scheduling Task Graphs with Contention in Communication

Figure 1: Tasks A, C_i, and B are scheduled for $T = 3$. For the two processor schedule two message passing tasks are scheduled on the network. The total execution time for both schedules is 5.

graph we frequently have to fork execution, creating multiple execution sequences where there was one to begin with, and then distribute those separate execution sequences onto different processing resources. If the ability to fork and distribute is not rate limited then we may create perfectly scalable applications. If we assume that there are rate limitations on moving data on and off processor, then there is a consequent limit on the rate of forking and distribution. Consequently, there will be a limit on our ability to exploit apparent concurrency.

Our example application shows the effects of limited I/O port capacity. We claim that other examples may demonstrate the effects of bottlenecks due to network capacity limits. In applications where the communications tasks are significant, i.e., there is a high CCR; communications resources will be heavily tasked. Unless one can realistically assume a network bandwidth that scales with the number of processors there is bound to be communication port and network resource contention. We cannot ignore communication resource contention; otherwise, we will obtain invalid schedules. We cannot ignore the limits on the ability to fork and distribute; otherwise, we will obtain schedules that tend to exploit apparent concurrency when the benefit is unobtainable.

A NEW APPROACH

To accurately predict mean execution time of a given application, and the maximum execution time to avoid failure of real-time applications, we need some detailed understanding of how message tasks are executed on a given architecture. The only way to minimize average execution time and not exceed a specified maximum execution time is to exert some control over the task schedule. If communication tasks are a significant cost in executing the task graph, they too must be controlled by being scheduled. Scheduling of message tasks requires that we can predict average and maximal times that a schedule will take to execute.

If we do not explicitly schedule, or queue, communication tasks then messages sent simultaneously and asynchronously may take maximal amounts of time to arrive completely. Secondly, if we can predict performance of individual tasks then we can predict message traffic and network loading. Finally, if we explicitly schedule message tasks, we can take advantage of our ability to predict network loading to reduce queuing-induced delays. We conclude that in order to obtain optimum performance we must schedule communications tasks. To do that we need to represent communications tasks so that they may be scheduled while simultaneously allowing for different models of network communication. A basic assumption here is that the task structure of communications may be made independent of the communications hardware. Below we introduce the message task constructs necessary for scheduling task graphs for execution onto message passing machines. We model the tasks represented by communications and define how they must be inserted into a task graph to be executed.

Representing message passing tasks:

There exist many different communications protocols depending on system designs and on the contexts of sender and receiver. Using DAG diagrams, we will describe three protocols below: the Synchronous Send, the Ready Send, and the Buffered Send. Each protocol specifies a definite sequence in which the Sending process, the receiving process, and the "network" commit to the passing of the message.

The Synchronous Send:

The Synchronous Send is illustrated in Fig. (2). Two sequences of task execution, beginning respectively with Task A and Task C, are executing on two processors. One task sequence initiates the sending of a message "X". The first communication is a message "envelope"

Synchronous Send

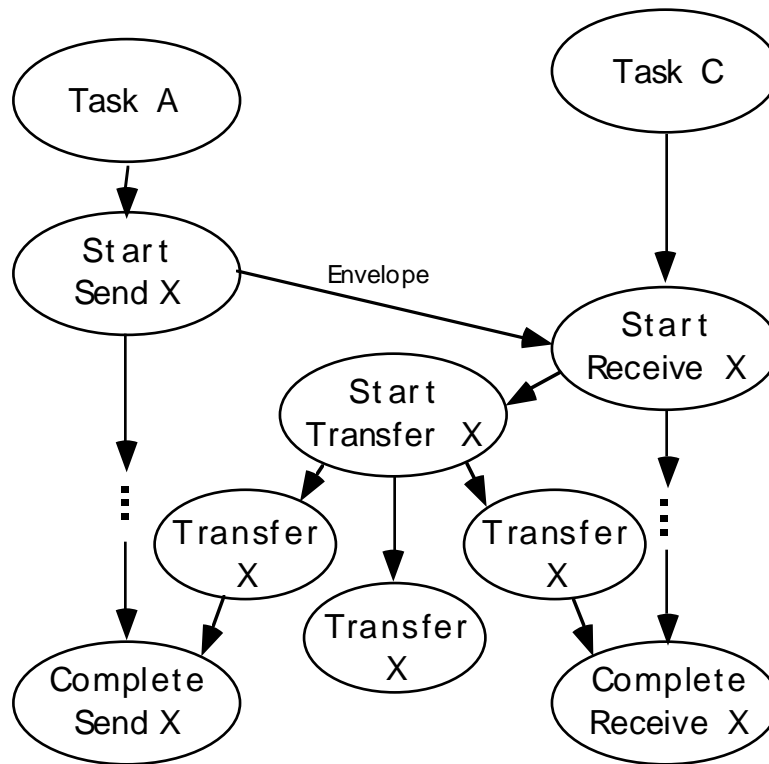


Figure 2.

which contains contextual information about the message to be sent, including, for example, the name of the variable (“X”) to be communicated and the size of the buffer required to receive the message. Sending this envelope signifies that the first processor is ready to transfer data. After this envelope is received, the second task sequence has sufficient information to allow the message transfer. In particular, the receiving task sequence may provide buffer space to receive the message. Upon completion of the Start Receive task on the second processor, both processors are ready for the data transfer. Finally, the network determines when the transfer actually takes place and synchronized data transfer processes occur on the first processor, the second processor, and the network. The three “Transfer X” tasks signify that actual data transfer occurs on three devices; the two processors and the network. Between the Start Send and Complete Send other tasks may be scheduled on the sending side under the assumption that the actual transfer, when it occurs, may occur asynchronously. If the transfer cannot occur asynchronously, then we assume that either the transfer task on the processor interrupts a processing task, or no other tasks will be scheduled between the Start Send task and the Complete Send task on the first processor. In any case, once the Start Send task is completed, access to the X buffer by the transfer process may not be blocked nor may writing occur until after the Complete Send task has finished. Similarly, on the receiving side, we assume that other tasks may be scheduled if there is no interference with the transfer process or the receiving buffer for X.

The Ready Send:

The Ready send is illustrated in Fig. (3). Two sequences of task execution, beginning respectively with Task A and Task C, are executing on two processors. The second of these task sequences indicates that it is prepared to receive a message, X, by sending a Ready signal. We may assume that a prior context exists whereby the receiving buffer can be properly sized for the safe receipt of the message. When the sending side has a message to send, Start Send is executed. Consequently, the transfer is enabled and occurs when the network is ready. As above, we assume that the actual data transfer may be asynchronous with other task processing on the communicating processors. Again, the three “Transfer X” tasks signify that actual data transfer occurs on three devices; the two processors and the network.

Ready Send

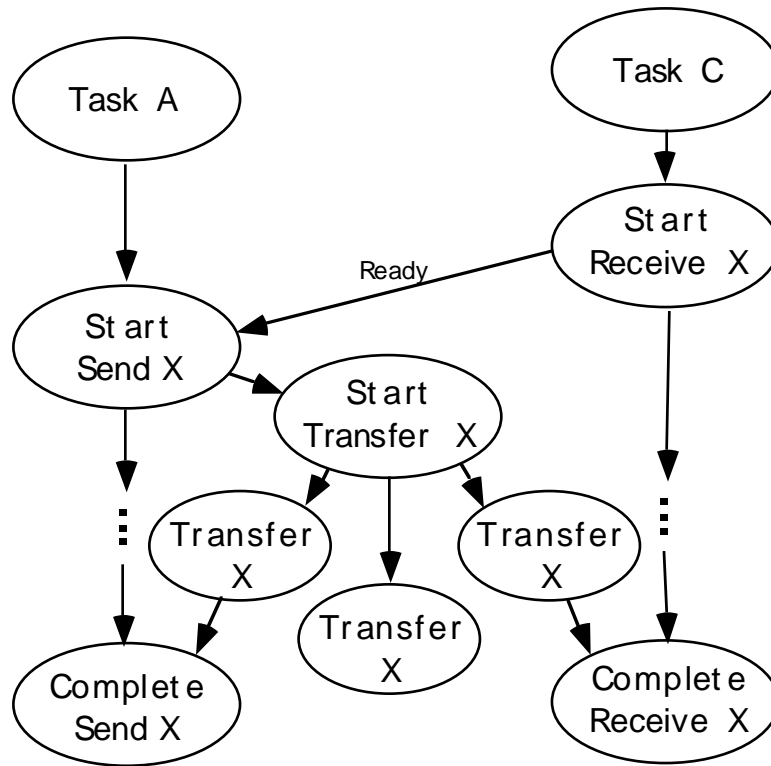


Figure 3.

The Buffered Send:

The Buffered Send is illustrated in Fig. (4). Two sequences of task execution, beginning respectively with Task A and Task C, are executing on two processors. One task sequence initiates the sending of a message “X”. The first communication is a message “envelope” which contains contextual information about the message to be sent, including, for example, the name of the variable (“X”) to be communicated and the size of the buffer required to receive the message. Sending this envelope signifies that the first processor is ready to transfer data. After this envelope is received, the second task sequence has sufficient information to allow the message transfer. If the receiving processor is not ready for immediate transfer then the message is immediately buffered. When the message to be sent is either transferred or buffered, the send may be completed, where completion means, as above, that the variable X may be reassigned by other tasks. (Note that the OR node of Fig. (4) is the only node that requires only one input to be enabled). Again, the three “Transfer X” tasks signify that actual data transfer occurs on three devices; the two processors and the network.

Buffered Send

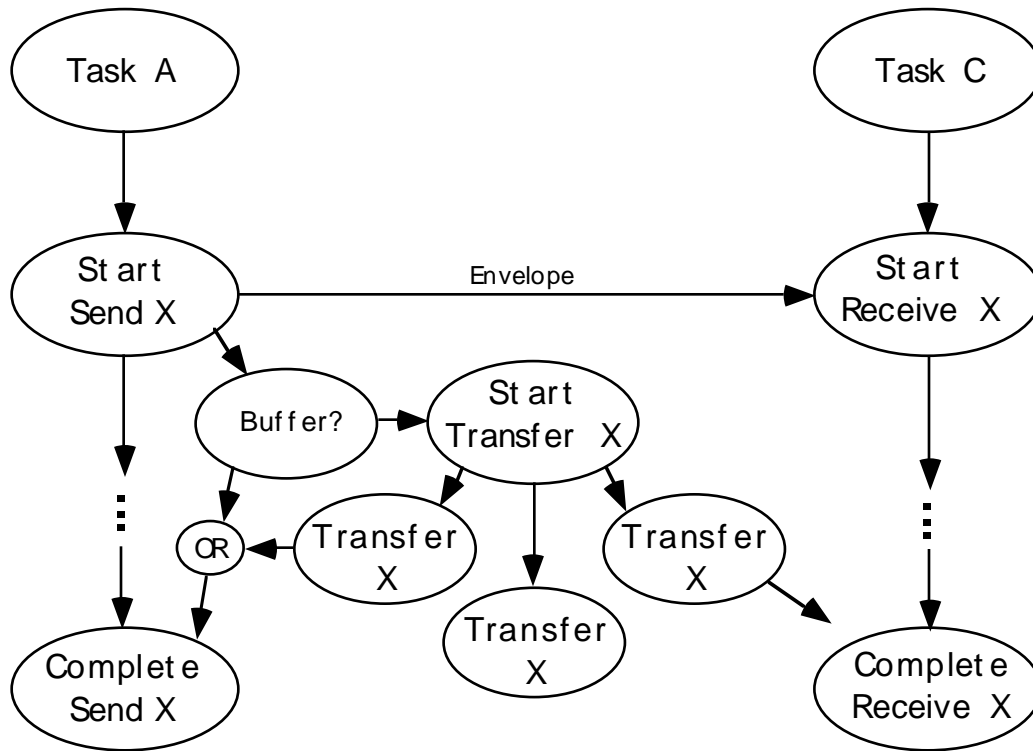


Figure 4.

HOW TO SCHEDULE A GRAPH WITH MESSAGE PASSING

We have introduced a messaging task structure that is independent of machine. The message sending task structures described above have two immediate effects: they change the task graph topology and they allow for the scheduling of the message sending tasks themselves. We have not yet discussed how one may schedule the task graph on any specific machine. The problem is that message passing or network architectures vary widely in their attributes and behavior. Therefore, we describe a way to do scheduling that is hardware independent and works for any specific machine. The way we do this is to define a scheduling method that uses a simulator to determine the behavior of the message passing hardware. Given any multiprocessor schedule of tasks that include message passing tasks and a simulator for a specific machine we can calculate the time to execute a given schedule. After we describe the interface to the simulator, we will discuss how to do scheduling using such a simulator.

THE NETWORK SIMULATOR

Because we need to calculate the cost of a schedule by simulation, the interface of the network simulator is specified to work together with other event-time simulation programs. Simulation of concurrent computational tasks without communication on a multiprocessor is straightforward. Communications may be handled by appropriately making calls to a network simulator. We want to notify the network simulator when message tasks are to be initiated. As the simulation of the execution of a schedule progresses in time, each time a message task is encountered on the schedule we need to call the network simulator. The network simulator will be called to simulate execution of message tasks. We also want the network simulator to notify us when a message task will be completed. We call the completion of a message task a network response event. The simulator will not be executed asynchronously as a co-routine. Instead, the network simulator will inform us of the time of occurrence of the next network-response event each time the network simulator is called. The network simulator will be called not only to queue message tasks but also to update the state of the network to a specified simulation time. Between calls, the network simulator will maintain all network state variables representing uncompleted message tasks and the network state resulting from the last simulation time update in an aggregate variable, `Network_State`. Each time the network simulator is called a new, current value for the next network-response event, `Next_Network_Event`, is returned, as is a new value for the variable `Network_State`. The variable `Next_Network_Event` also includes the time it occurs. A network response event is, in the context of the message sending protocols described, the start of any data transfer for a message and the completion of a data transfer.

The two calls to the network simulator will look like this:

```
(Next_Network_Event, Network_State)
    := Queue_Message_Task( Task, Start_Time, Max_Time, Network_State) and,
```

```
(Next_Network_Event, Network_State)
    := Update_Network_State( Next_Network_Event, Max_Time, Network_State).
```

`Queue_Message_Task` is used to task the network and `Update_Network_State` is used to advance the simulation time of the network simulator to the event time of its first input variable, `Next_Network_Event`. The network simulator will return a value for the next message-task completion-event dependent on the value of `Network_State` created by previous calls to the network simulator. To limit the amount of computation used by the network simulator, we indicate a maximum simulation time, `Max_Time`, beyond which we do not need to know about. If the network does not have a response event before `Max_Time`, then the

calls to the network simulator will return a null value, occurring at Max_Time, for Next_Network_Event.

The simulator will be used as follows. Some message tasks may be queued by calling Queue_Message_Task. A call to Update_Network_State establishes the time from which the network simulator cannot be reversed. Since Update_Network_State uses input possibly resulting from multiple calls to Queue_Message_Task, the calls to the network simulator should occur in time order according to the message task Start_Time. Each call to Queue_Message_Task will have a value for Start_Time that is greater than or equal to the previous value. There are two steps to calling the network simulator in time order. A set of message tasks should be queued in time order, one by one, by calling Queue_Message_Task. Secondly, there is at any time only one value for Next_Network_Event, and a corresponding time for that event. If between calls to the network simulator the corresponding time for Next_Network_Event is less than the Start_Time for the next message task to be queued, then Update_Network_State must be called before queuing any more message tasks.

OPTIMAL SEARCH, A*, AND HEURISTIC SEARCH

A “black-box” simulator can be used to create good schedules. An heuristic need not depend on the details of operation of specific network types. One of the most powerful methods for finding solutions to NP-complete problems, optimal search [7], uses the combination of an heuristic cost-evaluation function and an accurate cost-evaluation function. Below, we sketch out the approach for creating good schedules for executing task graphs on parallel processors. We use the notion of a partial schedule. A partial schedule is a truncated complete schedule. By successively appending ready-tasks to the ends of the individual processor schedule-sequences, we arrive at a complete schedule. A ready task is a task that has all of its predecessors finished. (We assume that communications tasks are inserted where they are necessary).

Algorithm A*:

Begin with an empty partial schedule.

With an heuristic, estimate the completion time, h_0 , the minimum time required to execute the task graph;

Associate the evaluation function value, $f_0 = h_0 = 0$, with the empty partial schedule;

Put the empty partial schedule onto a list called OPEN;

Loop:

Choose the partial schedule, index n , on the list OPEN having the least value, f_n . If the list OPEN is empty, exit with failure. If there are multiple partial schedules with the same minimal value of the evaluation function, choose a partial schedule that is actually a complete schedule, otherwise choose arbitrarily.

If p_n is a complete schedule, then accept it as the solution and exit Loop

Move the partial schedule, p_n , to a list called CLOSED.

Create all alternative partial schedules, $\{ p_i \}$ by adding one ready task to the partial schedule, p_n . A ready task is one with all predecessor tasks completed;

Using the simulator, compute the time, $\{ t_i \}$ to execute each newly generated partial schedule in $\{ p_i \}$;

With an heuristic, estimate completion times, $\{ h_i \}$, the minimum additional time required to execute the remaining unscheduled tasks for each newly generated partial schedule in $\{ p_i \}$;

Compute the evaluation function values, $\{ f_i \}$, where $f_i = t_i + h_i$ for each newly generated partial schedule in $\{ p_i \}$, and associate it with the corresponding partial schedule.

Put each newly generated partial schedule onto the list called OPEN;

End Loop

If the heuristic function, h , at all times returns a strict lower bound on the time to complete a partial schedule, then the solution schedule is guaranteed to be optimal. In the case that search times are too long using a given heuristic, various alternative schemes are possible as outlined in [7].

CONCLUSIONS

We have not, chosen at this time, an heuristic for using the optimal search approach. It has been our intent here to outline how to adequately deal with the problem of scheduling when there is contention in communications. We prefer to see scheduling results where computed schedule lengths are realistic reflections of task graph execution times on real hardware. We have demonstrated that it is necessary to separate the heuristic solution methods from an accurate evaluation of the time required to execute a specific schedule on a specific machine, and we have shown how to do this. It is not necessary to use the Heuristic Search method we suggested. Other approaches, e.g., [5], where integration of a simulation based computation of the objective function is feasible. We have specified realistic message passing protocols represented as task graphs that are not hardware specific. We outlined the requirements of a machine simulator for accurately computing the amount of time required to execute a specified schedule.

We believe that with our approach individual hardware vendors could provide simulators that allow others to compute how long it would take to execute specified schedules of tasks, including communications tasks. We expect that individual researchers can develop scheduling heuristics that can be tested for any machine and following our approach, it should be straightforward to determine which scheduling algorithms work best on any given machine.

References

1. A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman, LogGP: Incorporating Long Messages into the LogP Model - One step closer towards a realistic model for parallel computation, "7th Annual Symposium on Parallel Algorithms and Architecture SPAA'95", (July 1995).
2. D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken, LogP: Towards a Realistic Model of Parallel Computation, "Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming", (May 1993).
3. V. Kumar, A. Grama, A. Gupta, and G. Karypis, "Introduction to Parallel Computing: Design and Analysis of Algorithms", p.134, Benjamin/Cummings Publishing Company, Inc. (© 1994).
4. Y.-K. Kwok and I. Ahmad, Dynamic Critical Path Scheduling: An Effective Technique for Allocating Task Graphs to Multiprocessors, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 7, No. 5, (May 1996).
5. Y.-K. Kwok and I. Ahmad, Efficient Scheduling of Arbitrary Task Graphs to Multiprocessors Using a Parallel Genetic Algorithm, *Journal of Parallel and Distributed Computing* 47, 58-77 (1997).
6. C. E. Leiserson and B. M. Maggs, Communication-efficient Parallel Algorithms for Distributed Random-Access Machines, *Algorithmica*, Vol. 3, (1988) pp. 53-77.
7. N. Nilsson, "Problem-Solving Methods in Artificial Intelligence", p53, McGraw-Hill Book Company, (©1971) and also N. J. Nilsson, "Principles of Artificial Intelligence", p72, Springer-Verlag, (©1982).
8. G. C. Sih, "Multiprocessor Scheduling To Account for Interprocessor Communication", Memorandum No. UCB/ERL M91/29 (22 April 1991). Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720.
9. G.C.Sih and E.A.Lee, Dynamic-level scheduling for heterogeneous processor networks, "Second IEEE Symposium on Parallel and Distributed Processing", pp. 42-49, (1990).
- 10 J. D. Ullman, NP-Complete Scheduling Problems, *Journal of Computer and System Sciences* 10, 384-393 (1975).